Designing and Prototyping Data-Intensive Applications in the Logres and Algres Programming Environment

Filippo Cacace, Stefano Ceri, Letizia Tanca, and Stefano Crespi-Reghizzi

Abstract—This paper presents an environment and a methodology for the design and rapid prototyping of data-intensive software applications; i.e., applications which perform substantial retrieval and update activity on persistent data. In the proposed approach, the application is formally specified using Logres, a database language which combines object-oriented data modeling and rule-based programming. These specifications are translated into Algres, an extended relational algebra, thus yielding a rapid executable prototype. Finally, Algres programs embedded into a conventional programming language interface may be converted to conventional programs operating on a commercial relational system. This methodology helps automate the conversion from declarative requirements to imperative code, performing several tasks fully automatically and reducing the probability of human errors, while integrity constraints and application specifications are expressed in a declarative language, at a very high level of abstraction.

Index Terms—Deductive databases, object-oriented databases, software engineering, requirement analysis and design, rapid prototyping.

I. INTRODUCTION

The cost and complexity of large software projects has motivated research and experimentation on innovative environments for software development. In this paper, we describe an environment for the design and rapid prototyping of data-intensive software systems, developed at the Politecnico di Milano, in the context of the Algres and Logres projects [6], [11].

A. Logres and Algres

Logres and Algres are two advanced database programming languages.

• Logres is a new-generation database language which integrates an object oriented data model with rule-based programming, therefore belonging to the class of Deductive and Object-Oriented Databases (DOOD) [6]. Rules in Logres allow defining (possibly recursive) views and integrity constraints. Further, some rules may modify the database instance by inserting or deleting knowl-

edge—both facts and rules. An intuitive example giving the flavor of the Logres language is the following, where we define the class Person (with attributes name, age, and children; children is a set of persons) and the function Descendants, which returns the set of all the descendants of a person: Class Person = (name: string, children: {Person}, age:integer). Function Descendants: ( anc: Person ) -> descendant: {Person} ). Rules

Y \in \text{Descendant}(\text{anc} : X) 
\leftarrow \text{Person}(X, \text{children} : Z), Y \in Z.

Y \in \text{Descendant}(\text{anc} : X) 
\leftarrow \text{Person}(X, \text{children} : Z), W \in Z, Y \in \text{Descendant}(\text{anc} : W).

The Logres project started in 1989, with the first implementation of the Logres compiler, completed in early 1991, and a second one which is currently in progress.

• Algres is a relational system¹ for the management of complex (nested or ¬1NF) database relations, developed jointly with TXT Ingegneria Informatica SpA between 1985 and 1990, and is still evolving. The Algres language is an extended relational algebra, operating on complex structures built through the type constructors record, set, multiset, and sequence [2], [14], [23]; further, Algres supports a fixpoint operator (also called a closure operator [9]) to express recursive algebraic computations. As for Logres, we give an intuitive example of Algres by defining the relational object Person and then building, through an Algres expression, the set of all big families:

DEF Person: SET OF (name:string, age:integer, children: SET OF (child: name))

Big Family \leftarrow \text{SELECT } \{ \text{cardinality} \text{ (children) } > 4 \} \text{ Person}

Algres lacks programming constructs for implementing imperative computations. To gain expressive power and openness to external libraries, Algres was embedded into the C programming language, thus yielding the Alice

¹The reader is assumed to be familiar with the basics of application programming with relational systems.
of human errors. It is summarized as follows:

The Logres prototype is developed on top of the Algres system; indeed, the Logres compiler produces data definitions and programs in Algres that are subsequently executed by means of the Algres compiler and interpreter.

B. Rapid Prototyping and Design Methodologies

The software platform of Logres and Algres provides an environment for the rapid prototyping of data-intensive applications. By this term we mean any application which performs substantial retrieval and update activity on complex persistent data; therefore this term denotes a class that is broader than traditional database applications, and possibly includes programs that are usually written in procedural languages and supported by a conventional file system.

This environment is also used for the construction of operational systems running on conventional relational databases. To this purpose we have developed and experimented with a methodology which makes full use of the Logres and Algres compilers in order to semi-automatically generate a database and program design for a target relational environment, starting from high-level declarative specifications written in Logres.

The rapid prototyping environment and the software design method are by-products of our research on Algres and Logres, which were specified as innovative database languages and systems per se, regardless of their application to software engineering. While previous papers [11], [6] focused on the database perspective, in this paper we focus on the software engineering one. The positive experiences in software design, gained through the development of several case studies, have convinced us that these two applications of Logres/Algres are appropriate. The reader should notice that we use in full the compilation environment developed for Logres; indeed, the Logres compiler produces data definitions and programs in Algres that are subsequently executed by means of the Algres compiler and interpreter.

C. Summary of the Proposed Approach

The software-development methodology proposed in this paper allows one to automate most of the conversion from declarative requirements to imperative code, thus performing several tasks fully automatically and reducing the probability of human errors. It is summarized as follows:

- A high-level (object oriented) data schema is initially specified by means of the Logres-type system, which provides a notion of object identity, a number of type constructors, and part-of and isa links. Such a description may be incrementally modified, and mutual consistency of the various parts of the schema may be checked at compile time. Then, requirements and constraints for the application code being developed are formally specified by Logres rules.

- The specification is then given as input to the Logres compiler, which produces a version of it written in Algres by mapping the object-oriented schema to a -1NF relational schema, and mapping declarative statements to Algres algebraic expressions.

- Subsequently, the code written in Algres is given as input to the Algres compiler, which generates (normalized) flat relations.

- Finally, the output of this compilation process is considered by the designer, who should optimize and adapt the data structures of the program, and replace expressions in relational algebra with SQL queries so as to interface with the target relational system.

D. Outline

Section II presents the specifications of a case study, the ITS System. Through this case study we illustrate the features of Logres and demonstrate that deductive databases may model applications of great complexity. We also compare our style of collecting specifications with some alternatives.

We then concentrate on the transformation of the specifications. Section III presents an overview of our design methodology; Section IV describes the methodology for developing a rapid prototype from the specifications, and illustrates it on the case study. Section V describes the method for converting the prototype to a conventional relational application; this transformation is applied to the case study, which is thus completed. Section VI contains a critical discussion of the methodology, and Section VII draws our conclusions.

II. REQUIREMENT SPECIFICATION IN LOGRES

The case study proposed in this section is part of a larger application, which was developed by Agusta Sistemi, an aerospace company that builds helicopters, in the context of the Esprit Project 2443, Stretch [1], [7]. ITS consists of a technical part, containing a description of helicopters, and of a didactic part, containing information about the lessons taught to each student and his/her grades in the tests. The ITS system must be able to test the student’s ability to repair faults in the different parts of the system.

A. Database Specification

A Logres database is composed of an extensional part (the Extensional Database: EDB), corresponding to stored data (simple facts), and a (persistent) intensional part (the Intensional Database: IDB), consisting of rules that define the virtual data—essentially views; both the extensional and intensional parts are strongly typed.

The schema of the extensional database, shown in Fig. 1, models the technical description of the hydraulic system of a helicopter and the associated procedures for maintenance and trouble shooting. Each box represents a class in Logres. Solid arrows connecting boxes represent ISA hierarchies among classes, while dashed arrows represent part-of links. The intuitive meaning of the graphical representation is that, for instance, Maintenance Procedures are a specific subset of all
Fig. 1. Graphical representation of the database schema of the ITS application.

Procedures, and each Procedure refers to a specific Assembled-System.

Fig. 2 contains the type definitions for the Extensional Database displayed in Fig. 1. In particular,

1) The Domains Section contains the definition of types with no extent—ones for which the system does not maintain a set of current instances.

2) The Classes Section contains the definition of types of classes; each class has an associated collection of objects—its extent; each object has unique identification.

3) The Hierarchies Section contains the definition of isa relationships between pairs of classes, through definitions such as: C1 isa C2, whose interpretation is that each object of the class C1 also belongs to the class C2; hence the class C1 is a subset of the class C2, and operations defined for C2 (to be illustrated later) may be applied to objects of C1. Further, the type C1 is a structural refinement of the type C2 [8], [6], which means that all properties of C2 are also properties of C1.

4) A part-of relationship between two classes is established when one class occurs in the type definition of another class. The graph representing the part-of relationship among classes may be cyclic.

The type definitions for the data of the Intensional Database (IDB) are shown in Fig. 3(a) and (b). They also introduce further features of the Logres data model: associations and functions. The Logres data model, besides classes, also supports relations (called associations), thus integrating object-oriented and value-oriented concepts. While classes are sets of objects, associations are sets of tuples, arbitrarily complex.

In Fig. 3(a) we show the three intensional associations—link, chain, and cycle. The Associations Section defines their types, while the User Defined Rules Section contains the corresponding rules. Associations describe the topology of the hydraulic system:

- Link contains pairs of directly linked systems.
- Chain contains minimal paths among any two systems, and additional information about the path length.
- Cycle contains minimal cyclic paths.

The extension of the above associations is evaluated by Logres Rules, whose general structure is: L ← L1, ..., Ln, where L, L1, ..., Ln represent positive and negated predicate occurrences. The Logres rule language is a typed extension of Datalog [10], intended to be computationally complete both for queries and updates. Features of the language include negation in the body and in the head of rules, complex variables, and the creation of new objects through the invention of identifiers (see also [3], [6]). Negative predicate occurrences in the rule heads are interpreted as tuple deletions.

Observe that the recursive rule defining the Chain association exhibits several of the advanced features of the rule language, including sets and negation in the rule’s body in order to prevent the derivation of infinite paths.

The Logres data model also allows the definition of set-valued functions. Functions are used for collecting sets of objects with given properties; each function in Fig. 3(b) is defined by one or more set membership rules:

- The function Bill-of-mat computes recursively the set of components of each structured system.
- The function Sys-proc computes the set of maintenance procedures which apply to each system.
- The function Repairs computes the set of procedures repairing a given symptom.

Logres can also express integrity constraints in a declarative way, in the form of denials; when their body, normally false, becomes true, then a transaction is rolled back. The two consistency constraints, included in the User-Defined Rules Section shown in Fig. 3(c), have the following meaning:

- No system can be a subcomponent of itself.
- For any two systems S1 and S2, if S1 is adjacent to S2, then S2 is adjacent to S1.

In Logres, constraint violations can be easily transformed into the raising of exceptions. For instance, by adding the rule heads error (selfsubsystem, S) and error (adjacency, S1, S2), respectively, constraint violation would raise two exceptions.

The language allows one to resolve possible ambiguities between the definitions of isa and part-of relationships.2

2In our example, classes are extensional and associations are derived. This is not a mandatory feature of the Logres language: some classes could be intensionally defined, and some associations could be extensional.
**DOMAINS SECTION**

- tools = (tool: string); steps = <step: string>;

**CLASSES SECTION**

- system = (part-n: integer, serial-n: integer, system-t: integer, system-n: string);
- prod-system = (system, materials: material), prod-brand, model: string, month: integer, year: integer, lasting: integer);
- material = (mat-name: string, price: integer);
- brand = (brand-name: string);
- struct-system = (system, components: comp: system), linked-systems: link-type: string, direction: string, sys: system);
- assembled-system = (prod-system, struct-system, surr-el: location: string, temp-min: integer, temp-max: integer);
- procedure = (appl: assembled-sys, name: string, tools, manual-chap: string, time: integer, steps);
- maint-proc = (procedure, in-the-case: string);
- trouble-shoot = (procedure, symptoms: symp: string, repairing: cause: string, isolation: string, remedy: string);

**HIERARCHIES SECTION**

prod-system ISA system
struct-system ISA system
assembled-sys ISA prod-system, struct-system
maint-proc ISA procedure
trouble-shoot ISA procedure

Fig. 2. Schema of the Extensional Database (EDB) of the ITS system, written in Logres.

**ASSOCIATIONS SECTION**

- link = (from: system, to: system);
- chain = (from: system, to: system, leng: integer);
- cycle = (system, leng: integer);
- link(from: S1, to: S2) \< struct-system(system: S1, linked-systems: L),
  link-type: "tube", direction: "out", sys: S2 \> L.
- chain(from: S1, to: S2, leng: D) \< link(from: S1, to: S2, D = 1.
- chain(from: S1, to: S3, leng: D1) \< chain(from: S1, to: S2, leng: D), link(from: S2, to: S3),
  D1 = D + 1, NOT chain(from: S1, to: S3).
- cycle(system: X, leng: D) \< chain(from: X, to: Y, leng: D), X = Y.

**FUNCTIONS SECTION**

- bill-of-mat: system \-> (system)
  SUB bill-of-mat(system: S) \-> struct-system(system: S, components: C), SUB C.
  SUB bill-of-mat(system: S) \-> S \< bill-of-mat(system: S),
  struct-system(system: S, components: C), SUB C.
  sys-proc: system \-> (procedure)
  PE sys-proc(system: S) \< procedure(P, appl: S).
  repairs: (symptom: string) \-> (procedure)
  PE repairs(symptom: S) \< trouble-shoot(P, symptoms: S1), SE S1.

**USER DEFINED RULES SECTION**

- \< system(S), S \< bill-of-mat(system: S).
- \< assembled-sys(S1, surr-el: SE1), assembled-sys(S2, surr-el: SE2), S1 \< SE2, NOT(S2 \< SE1).

Fig. 3. Declarative specifications of the Intensional Database (IDB) of the ITS system: (a) associations and their rules; (b) functions and their rules; and (c) user-specified integrity constraints.

by generating the facts error(selfsubsystem,s) and error(adjacency,s1,s2), where s, s1, s2 are suitable constants. Exception management could then be programmed by rules, which are activated when errors occur; alternatively, it could be programmed by designers during the prototype conversion phase.

In addition to the user-defined consistency constraints, the Logres system automatically derives referential integrity constraints from the schema. There is one such constraint for each isa and part-of relationship, and one for each reference from an association to a class. Constraints are expressed as rules and added to the IDB.

In Fig. 4 we show the referential constraints corresponding to the class assembled-sys and to the association link. Note that these constraints implement a specific policy for dealing with referential integrity, which treats classes and associations in a different way:
**SYSTEM-DEFINED RULES SECTION**

\[
\begin{align*}
prod-system (self:X) & \leftarrow \text{assembled-sys (self:X)}. \\
struct-system (self:X) & \leftarrow \text{assembled-sys (self:X)}. \\
system (S) & \leftarrow \text{assembled-sys (surv-el:SE)}, \; S \in SE. \\
\text{assembled-sys (S)} & \leftarrow \text{procedure (P, appl: S)}. \\
\text{NOT (link (L))} & \leftarrow \text{link (L, from:S1, to:S2, NOT (system (S1)))}. \\
\text{NOT (link (L))} & \leftarrow \text{link (L, from:S1, to:S2, NOT (system (S2)))}.
\end{align*}
\]

Fig. 4. Active integrity constraints derived by the system from the schema of the ITS database.

- When the object of a subclass is inserted, the correctness of an isa hierarchy is preserved by adding that object into its superclasses. Thus when an object is introduced in the assembled-sys class, it is automatically inserted into the system class. Instead, the deletion of an object from the assembled-sys class does not cause any automatic change. To delete completely a real-world object from the database, it is necessary to delete the various objects which exist in the generalization hierarchy, starting from the leaves and proceeding toward the root class of the hierarchy.
- All tuples of associations must refer to existing objects. Thus when one object is deleted from the class system, any instance of the link association referring to that object is also automatically deleted.

In summary, the Intensional Database (IDB) of the ITS system consists of recursive and nonrecursive rules that

1) Recursively derive the components of each structured part.
2) Infer information about how the different parts of the hydraulic system are linked to each other.
3) Impose some consistency constraints on the instance.

Note that the Intensional Database is persistent and application independent.

**B. Specification of the Applications**

Application programs are expressed through modules. A module in Logres is a triple, including a set of rules, a set of type declarations, and a goal; each of these terms may be missing. When a module is applied to a database, it may change its content by affecting either the extensional database or the intensional database, or both. The capability of expressing updates through rules is one of Logres' design goals and makes it suitable as a language for specifying general-application programs. The execution of a module is qualified by an option, which determines the persistence of its side effects. The option has six possible values:

- **RIDI–Rule Invariant Data Invariant**: Module execution corresponds to an ordinary query and does not alter the extensional database.
- **RADI–Rule Addition Data Invariant**: Module execution adds new rules to the persistent intensional database and does not alter the extensional database.
- **RDDI–Rule Deletion Data Invariant**: Module execution deletes rules from the persistent intensional database and does not alter the extensional database.
- **RIDV Rule Invariant Data Variant**: Module execution updates the extensional database, without changing the rules of the persistent intensional database.
- **RADV–Rule Addition Data Variant**: Module execution updates the extensional database and adds new rules to the persistent intensional database.
- **RDDV Rule Deletion Data Variant**: Module execution updates the extensional database and deletes rules from the persistent intensional database.

We consider two example applications—the former computes views and queries, the latter computes an update.

1) **Application 1**: The first application is a query module which must be able to answer, for all subsystems of a given one, the following queries:

1) Find the average repair time for a given symptom.
2) Find the maximum repair time for a given symptom.
3) Find the minimum repair time for a given symptom.
4) Find the minimum of the maximum working temperature of all subsystems of a given system.

The Logres program for the query module is shown in Fig. 5. The module is executed with the RIDI qualification and does not change the database. The four rules of the Rules section correspond to the above four queries. The parameter option in the goal allows the user to indicate which query should be performed.

For example, to find out which is the average repair time for a loss of power in the pump whose part name is 2512, the query is ? - query-mgr (input (system (part-n:2512, name: "PUMP"), symptom: "power loss"), output (time: T), option: 1).

The Logres program for the query module is shown in Fig. 5. The module is executed with the RIDI qualification and does not change the database. The four rules of the Rules section correspond to the above four queries. The parameter option in the goal allows the user to indicate which query should be performed.

For example, to find out which is the average repair time for a loss of power in the pump whose part name is 2512, the query is ? - query-mgr (input (system (part-n:2512, name: "PUMP"), symptom: "power loss"), output (time: T), option: 1).

The first three rules use the repairs function, defined in the application-independent part of the IDB, and the built-in functions avg, max, and min, respectively; instead, the fourth rule uses the function tempmax, which is defined within the module and does not belong to the IDB. After applying the module, tempmax is discarded; if instead the module were applied with a RADI option, then the four rules for query – mgr and the function tempmax would become persistent.

2) **Application 2**: The second application performs an update by substituting the pumps installed in all subsystems subject to electric faults with other pumps described in the class New. The program, illustrated in Fig. 6, has a query part and an update part. The former is the rule faulty-sys which identifies all faulty systems; the latter consists of the two rules for struct-system, which operate on them. The Logres module must be qualified with the option RIDV so that the changes to the EDB are made persistent, while the IDB does not change after module execution.

The function pumps has no input parameters and is used to build the set of objects in system that are pumps. Note that
functions with no arguments are used to group elements into named sets: the function `pump` can be considered as a sort of intensionally defined subset of `system`.

The last two rules substitute the old pumps with the new one: note that it is necessary to propagate the pump substitution to the `part-of` attribute `linked-systems`, which is composed by elements of the class `system`. Thus two update rules are needed. Finally, notice that the new pump is not explicitly inserted into the class `system`: the referential integrity constraint generated by the ISA declaration `struct-system ISA system` of Fig. 2 will include it automatically.

**C. Logres as a Specification Language**

We compare Logres as a requirement specification language with other specification techniques. A comparison of Logres with other deductive databases can be found in [6].

In comparison with formal algebraic specification methods proposed by the software engineering community, Logres has several advantages in our opinion. We have used RAP [16], a classic algebraic specification language, for formally specifying the Algres system itself (see [20]), but this experience was not very successful; the resulting prototype was extremely slow and its structure was so distant from the actual compiler and interpreter that it was not suitable for evolutionary prototyping or conversion to conventional programming languages; this limitation would be even more severe in a conversion to conventional database languages and systems. We also note that all methods which stress
the use of abstract data types deliberately choose to hide the data structures being developed and to concentrate on methods' functionalities, while in data-intensive applications supported by conventional databases it is important to expose and carefully design such data structures (schemas) as part of the designer-controlled interaction. On the other hand, ADT supporters claim that they are able to automatically prove properties of their solutions, while Logres does not provide this possibility.

If we consider instead approaches that were developed in the information-systems community, then we find several similar approaches; DAIDA, reported in [12] and [5], is the most significant. With respect to the architecture of DAIDA, it is worth noticing the following differences:

- DAIDA operates at three levels: the World's Model Level, the Conceptual Design Level, and the Database Programming Level. We consider only the second and third levels. In other words, we do not consider requirements in their most abstract form. Logres should therefore be compared to the Taxis Design Language (TDL) [18], rather than to the Requirement Modeling Language (RML) [17].
- DAIDA has an advanced database programming language as target, while we consider a conventional (relational) database system as target; thus programs generated in the DAIDA environment can express a much richer semantics, although for specific target systems.

TDL is a first-order predicate language, combined with object orientation; it has several properties in common with Logres, including the fact that it can generate a rapid functional prototype in Prolog [12]. The main difference between Logres and TDL is that TDL has a pre/postcondition style to express specification of transactions in terms of the states before and after transaction execution, while Logres describes transactions explicitly in terms of modification actions that change the database state. Thus TDL is more declarative than Logres. Note that rules have a fully declarative semantics, but the imperative part of Logres resides in fixing the sequence and modes of module applications. Logres specifications, however, may be immediately translated into executable code by a compiler which performs fairly standard transformations, well-defined in the world of deductive databases, which will be explained in the next sections.

Our methodological approach does not currently focus on requirements modeling. For this task we can take advantage of a very rich literature and some specific proposals. Among them, we recall:

- The body of knowledge that has been developed on the Entity-Relationship Approach to database design. This field is subject to commercial evolution which sometimes lacks solid methodological background; however, recent proposals assume versions of the data model which incorporate several object-oriented concepts and deal with user requirements in a structured and disciplined way [4].
- Object-oriented database-design methodologies, such as the Ithaca methodology [13]. Ithaca proposes a rich object-oriented model with objects, roles, states, properties, messages, and rules. The requirement-analysis phase describes how to derive specifications in the above object-oriented model, starting from real-world requirements.
- The Requirement Modeling Language of DAIDA [17], a specification language for describing evolving information systems; this language, in particular, proposes situational calculus to deal with temporal events which are currently not considered by Logres.

The next two sections illustrate how the Algres-Logres programming environment may be effectively used to translate declarative specifications into imperative programs.

III. METHODOLOGY

This section briefly overviews the software design methodology proposed in this paper, described in Fig. 7. Declarative specifications, shown in the upper-left corner of Fig. 7, consist of an application-independent portion, common to all applications, and several modules, one for each application.

The Logres compiler translates applications from Logres into Algres. This translation transforms declarative logic-based programs into programs in an extended relational algebra, where the algebraic expression dictates a specific order of execution of relational operations; we regard these programs as semi-imperative. The Logres compiler is responsible of translating, once and for all applications, the Logres schema of the EDB into an Algres schema. Further, the Logres compiler is invoked for each Logres module and generates appropriate applications, written in Alice.

The Algres compiler is subsequently applied to both the schema and the applications; data structures in Algres are translated into flat relations in RA (relational algebra), and the Algres expressions are translated to conventional algebraic expressions in RA. Finally, Alice applications are compiled by a conventional C compiler; the executable code includes calls to the interpreter of RA expressions. These transformations, shown in the right part of Fig. 7, produce a prototype of the applications, which operates on flat relations and is written in C and algebraic expressions. Run-time efficiency of the prototype is comparable to that of other interpreted systems.

The Algres program and RA schema constitute the best starting point for a further (manual) transformation that generates a program operating on a conventional relational platform by progressive and localized transformations of the prototype code through an evolutionary process; this is shown in the bottom left part of Fig. 7.

Our specifications include in the application-independent declarations a very large fraction of application semantics in the form of rules describing views and constraints. In contrast, conventional relational databases are semantically inexpressive, since their schema definitions only describe flat data structures and a limited collection of fixed-format integrity constraints. This relationship is described graphically in Fig. 7: in the Logres corner, modules (i.e., application programs) have a small surface compared to the combination
of the general schema and rule definitions; instead, application programs produced at the end of the design are large, while the final relational schema is small. This shows that some of the semantics modeled through generic rules and schema definitions in Logres, which cannot be supported by relational schemas, has migrated into application programs without being lost.

IV. DEVELOPMENT OF A RAPID PROTOTYPE

We now present each design step in detail, and exemplify it through the case study.

A. Logres Compilation

The Logres compiler performs the following transformations:

- **Schema translation.** Schema definitions in Logres are translated into schema definitions in Algres. The following transformations are performed by the compiler:
  - Object identity of Logres classes is provided in Algres by generating new attributes, named Self, as tuple identifiers.
  - Generalization hierarchies among Logres classes are modeled by generating one Algres object for each Logres class; all Algres objects which are related through an ISA hierarchy (e.g., a Student who is also a Person) have the same numeric value in the Self attribute. This translation scheme is similar to what has been proposed for the implementation of OODB’s on relational platforms [15]; it enables the reconstruction of a generalization hierarchy through equijoins on the Self attributes; joins are automatically generated by the Logres compiler.

Fig. 8 shows a portion of the Algres schema produced by translating the EDB Schema of Logres. This example shows the static features of the Algres language, an extended relational language supporting complex relations. Objects are built through orthogonal-type constructors, including records, sets, bags, and sequences (that is, lists), but there is no semantic complexity (no object identification and isa hierarchies).

- **Module translation.** Each Logres application module is translated into a corresponding Alice module. The Logres compiler extracts rules which are relevant to the module from the general application-independent rule set and applies significant optimizations to them.

The process of rule extraction is recursive; for all the
DEF PERSISTENT system: SEQUENCE OF (oid: INTEGER, part-n: INTEGER, system-t: INTEGER, system-n: STRING)
DEF PERSISTENT material: SEQUENCE OF (oid: INTEGER, mat-name: STRING, price: INTEGER)
DEF PERSISTENT brand: SEQUENCE OF (oid: INTEGER, brand-name: STRING)
DEF PERSISTENT assembled-sys: SEQUENCE OF (oid: INTEGER, surr-el: SET OF (system: INTEGER), location: STRING, temp-min: INTEGER, temp-max: INTEGER)
DEF PERSISTENT maint-proc: SEQUENCE OF (oid: INTEGER, in-the-case: STRING)

Fig. 8. Algres data structures for the Extensional Database of the ITS system.

rules in the module, their RHS is examined, and all the rules corresponding to predicates appearing in the RHS are extracted; in turn, their RHS is examined and possibly other rules are recursively extracted. Note that this process not only derives rules defining the IDB and goal computations, but also extracts rules which support integrity constraints, which are consequently enforced by all applications. Rule extraction for Application 2 is shown in Fig. 9.

All the extracted rules are progressively translated into expressions, written in Algres-Prefix. Four different optimizations are performed:

1. Determining the rule evaluation order. The Logres compiler identifies mutually recursive rules and clusters them into sets (also called strong components), then determines the order of evaluation among these sets.
2. Determining, within a rule, the predicate evaluation order. This problem is very similar to conjunctive query optimization. In general, the most instantiated predicates are evaluated first (e.g., those which can take advantage of the selectivity due to query constants).
3. Rewriting the rules so as to make their computation more efficient. In particular, the Logres compiler considers an algebraic variant of the magic set optimization in order to reflect the selective conditions written in the user’s goal.
4. Finally, with recursive queries, improving fixpoint computation, taking advantage of special cases (e.g., linear rules).

These optimizations are significant and would otherwise be left to the application programmer. They transform highly declarative program specifications into semi-imperative algebraic expressions.

Fig. 10 shows, for Application 2, the Algres expressions corresponding to the predicates pumps, faulty - sys, and struct - sys of Module 2. Conjunctions of Logres classes or associations in the body of a rule are translated into joins; joins are also necessary to navigate along part-of and isa links among classes. The third expression contains the Algres UPDATE operator, which is used to update the value of the attribute components. For an explanation of each individual operation, see [11].

Fig. 11 shows the general schema of fixpoint computation.

B. Algres Compilation

The next transformations are due to the Algres compiler.

- Schema translation. The Algres Compiler generates, from schema definitions in Algres, a schema which only uses flat tables. This is achieved by defining one table for each type constructor which appears in the definition of an Algres object. Therefore each Algres object is
CACACE et al.: DESIGNING AND PROTOTYPING DATA-INTENSIVE APPLICATIONS

PUMPS ← SELECT [system-n = "pump"] SYSTEM
MERGE JOIN [oid] PROJECT [oid, assembled-sys] procedure
SELECT [exist (cause = "electric fault") repairing] TROUBLE-SHOOT
STRUCT-SYS ← UPDATE [components := UNION NEW DIFFERENCE COMPONENTS PUMPS] STRUCT-SYS

Fig. 10. Example of the translation of Logres rules of Application 2 into Algres expressions.

appl3 ()
{ int condition;

Evaluation of Algres expressions

Union of the facts derived at the present iteration with the old ones.

Evaluation of termination condition

GETOBJ COND via (condition)
return condition;
}

main 0
{ ALGRES [graphic]; /* activation of ALGRES run-time support
def persistent SYSTEM: <oid:integer, part-n: integer, ... >
def persistent PROD-SYS: <oid:integer, materials: (material), ... >
def persistent MATERIAL: <oid:integer, material-name: string, ... >
... /* definition of the objects of the part of the ALGRES schema
/* used in the application
while (! condition)
condition := appl3(); /* iterates the evaluation of rules until saturation
ENDALGRES;
}

Fig. 11. Alice program generated by the Logres compiler starting from the program of Application 2.

mapped to several flat tables; they are connected by specific attributes, called, respectively, Pointer and Pointed, that implement one-to-one or one-to-many relationships. Further, multisets and sequences are modeled by adding suitable attributes to flat tables: Occ, expressing in Algres multisets the number of object occurrences with the same tuple value, and Pos, giving the relative position of Algres tuples within sequences. Flat tables are then implemented in RA, the Relational Algebra interface. Fig. 12 shows some of the relations generated according to the above methodology for the ITS example.

- **Module translation.** The Algres code is translated into instructions in RA; namely, to specific algebraic operations on flat main-memory tables such as selections, projections, joins, and duplicate eliminations. Instructions are embedded into C programs, already present in Alice, which are not modified by this translation.

**C. Execution of the Prototype**

The execution of the rapid prototype requires compiling the C program obtained at the end of the above transformation; RA instructions are translated into calls to the RA interpreter, which executes each RA operation in sequence in main memory. The current prototype is single-user and not persistent. However, it allows a full simulation of applications: test data can be provided to populate the database, and applications may be tested by making full use of the Algres programming environment that includes graphical user interfaces for data input, schema browsing and interactive querying; it currently runs on X-Windows and Motif, on Sun/Sparc Workstations, or on the PC-OS2 Presentation Manager.

One of the components of the Algres programming environment is a loose-coupling mechanism for loading/storing data from/to persistent, relational databases (currently Informix). This tool is used to generate standard SQL definitions; at this point the prototype can be tried on a conventional database, although with a loose-coupling approach: data are retrieved at the beginning of the session and stored back at the end. In this way it becomes possible to simulate the evolution of the database through several design sessions.

**V. CONVERSION OF THE PROTOTYPE**

The last set of transformations, discussed in this section, produces a relational implementation.

- **Schema Transformation.** The relational schema is produced by the Algres Compiler, as discussed in the previous subsection; it includes suitable indexes, some of which are unique (for instance, on the Self, Pos, and Pointer attributes). Additional manual transformations may be performed in order to improve the schema; for instance, by dropping some of the artificial attributes (e.g., Self attributes when the table has a natural key attribute), adding or dropping indexes, and possibly applying denormalization of relations. Finally, some integrity constraints such as key uniqueness and referential integrity that are directly supported by (some) relational systems are added to the schema definition.

- **Program Transformation.** This transformation step assumes the description of a module in Alice, disregarding the transformation of Algres expressions into RA instructions which are needed in the rapid prototype. It includes, however, the loosely coupled interface with the external relational system. The generation of efficient code proceeds as follows:

1. The programmer should define the main-memory data structures which are most efficient. A typical transformation consists in replacing the attributes Pointer and Pointed by physical pointers. These data structures are defined through C types.
RA-prod-sys1 = (POINTED, material: INTEGER)

Fig. 12. Example of RA data structures for the Extensional Database of the ITS system.

CREATE TABLE SYSTEM
(SELF INT NOT NULL,
PART-N CHAR(8),
SERIAL-N CHAR(12),
SYSTEM-T VARCHAR(10),
SYSTEM-N VARCHAR(20),
PRIMARY KEY SELF)
CREATE UNIQUE INDEX SYS-ID ON SYSTEM (SELF)
CREATE TABLE STRUCT-SYSTEM (SELF INT NOT NULL)
CREATE UNIQUE INDEX STRUCT-SYS-ID ON STRUCT-SYSTEM (SELF)
CREATE TABLE STRUCT-SYSTEM-COMP
(SYSTEM INT NOT NULL, 
COMP INT NOT NULL)
CREATE UNIQUE INDEX STRUCT-SYSTEM-SYSTEM ON STRUCT-SYSTEM-COMP (SYSTEM)
CREATE UNIQUE INDEX STRUCT-SYSTEM-COMPONENTS ON STRUCT-SYSTEM-COMP (COMP)

Fig. 13. SQL instructions for the relational schema of ITS.

2. Next, the programmer should reprogram the Algres part of Alice programs by using C data structures in main memory, thus exploiting the new physical structures specifically available.

3. Finally, the programmer should reconsider the import and export instructions, which are already mapped to SQL queries and implemented through cursors. This loosely coupled interface should be transformed into a tightly coupled interface, especially if the application is highly interactive.

To understand the rationale of the third transformation, consider an application which assumes input from the user and then performs a query or update on a specific object, selected through the assumed input value; in this case, the standard Algres translation would first read all object instances in main memory, then operate a main-memory selection returning the specific object. A tightly coupled interface, instead, performs the selection directly on the database; a similar transformation may be performed on the rapid prototype as well if the efficiency of the prototype is important. It should be noted that the loosely coupled interface provides a general template for the SQL cursor, whose modification is rather straightforward.

Fig. 13 shows part of the relational schema generated for the ITS database. Note that each original complex attribute in Logres or Algres (for instance, the attribute components) requires an additional relation, which is indexed on both key attributes. Indexes are also added for the self attribute on each relation corresponding to a class.

VI. EVALUATION OF THE PROPOSED APPROACH

The design methodology proposed in this paper has several advantages and some disadvantages. On the positive side:

1) Logres describes global, application-independent knowledge in the form of structurally and semantically complex schema declarations, flexible intensional database, and generalized integrity-preserving rules. It differs from first-order logic, which is commonly used for modeling information systems, because of its clausal form, and because it adds extra-logical features such as strong typing, set constructors and set predicates built in the language, and rules producing side effects on the extensional database, such as insertions and deletions.

2) Logres specifications are fully declarative. Schema definitions in Logres abstract from the physical relational implementation which is eventually produced. The semantics of rules depends neither on the order of rules nor on the relative order of predicates within rules.

3) A number of difficult optimizations are performed by the Logres and Algres compilers; in particular, they transform declarative order-independent specifications into imperative order-dependent specifications by choosing the optimal order of rule evaluations and of predicate evaluations within rules.

4) A rapid and reasonably efficient prototype is immediately available, allowing the simulation of the final system on a small main-memory implementation, in order to evaluate how the specified system reflects users' requirements.

However, the proposed approach presents several problems:

1) The approach is valid only for applications which are globally developed, top-down, from scratch. The evolution of the operational database is not under direct control, and in particular, independently developed applications may violate the integrity constraints, which are instead supported by all the applications developed in the context of the methodology.

2) Though Logres is fully declarative, it imposes some restrictions due to particular choices in the language semantics. For instance, we generate rules from schema
declarations that enforce particular policies concerning referential integrity among classes and associations. Similarly, Logres uses a particular type of semantics, called inflationary semantics [19], for determining the meaning of particularly difficult rule sets; this is one of many possible choices.

3) Logicians programming is a difficult programming discipline, not particularly popular among database programmers. This problem can be partially eased by developing different syntactic presentations of Logres schema definitions and rules.

VII. CONCLUSIONS

This paper has presented two main contributions. First, we have demonstrated the use of Logres as a specification language for data-intensive applications. Second, we have described a design methodology, largely based on the Algres/Logres programming environment, for the progressive transformation of declarative specifications into operational systems which use relational technology.

We assume the requirements to be best represented in a declarative manner, where they are at the maximal level of abstraction, independent from the hardware and software platforms being chosen. We believe Logres provides the right abstractions for "reasoning about requirements," while the Algres/Logres programming environment provides very powerful mechanisms to translate this knowledge into conventional programs.

ACKNOWLEDGMENT

The authors wish to thank G. Lamperti from TXT, the scientists of Agusta Sistemi who contributed the ITS case study, and all the students who have worked with us.

REFERENCES

Letizia Tanca graduated in mathematics from the University of Naples, and received the Ph.D. degree in 1987. She is an Associate Professor in the Dipartimento di Elettronica e Informazione of the Politecnico di Milano. Her main research interests concern deductive databases, optimization of recursive queries, database programming languages, applications of logic to computer science, and graph-based database languages. She is a co-author of the book, Logic Programming and Databases (Springer-Verlag, 1991).

Dr. Tanca is a member of the Publishing Committee of the Rivista d'Informatica, the journal of the Italian Association for Computer Science, and is a member of the ACM.

Stefano Crespi-Reghizzi graduated in electrical engineering from the Politecnico di Milano, and received the Ph.D. degree in computer science in 1970 from the University of California, Los Angeles. He was previously with the Università di Pisa. He is now a Professor of Computer Science in the Dipartimento di Elettronica e Informazione at the Politecnico di Milano, and is the Coordinator of the doctoral program in computer and automation engineering. His research interests focus on the convergence of software engineering and database technologies. He is also engaged in the theoretical investigation of formal languages, automata, and semantics.